L9: Entry 11 of 14 File: USPT Mar 12, 2002

DOCUMENT-IDENTIFIER: US 6356915 B1
TITLE: Installable file system having virtual file system drive, virtual device driver, and virtual disks

Brief Summary Text (20):
One solution to these problems is to create and install a custom file system which replaces or otherwise augments the pre-existing or "native" file system. Such a custom file system could be designed to allow freer reconfiguration, and to allow storage and display of more file information. However, implementing the hundreds of function routines included in a file system is an exceptionally complex task. If successful, the resulting custom file system would be an undesirably large and complex program which would be difficult to successfully debug, and cost an excessive amount to create.

L9: Entry 10 of 14 File: USPT Mar 26, 2002

DOCUMENT-IDENTIFIER: US 6363400 B1
TITLE: Name space extension for an operating system

Brief Summary Text (20):
One solution to these problems is to create and install a custom file system which replaces or otherwise augments the pre-existing or "native" file system. Such a custom file system could be designed to allow freer reconfiguration, and to allow storage and display of more file information. However, implementing the hundreds of function routines included in a file system is an exceptionally complex task. If successful, the resulting custom file system would be an undesirably large and complex program which would be difficult to successfully debug, and cost an excessive amount to create.

CLAIMS:

3. The method of representing data as files in a data processing system of claim 2, wherein the data repository system comprises a version control system and the custom file attributes include, for each of the files in the second portion: a status of the file, if a user has gained exclusive control of the file, a name of the user, and a revision number of the file.

L9: Entry 1 of 14 File: PGPB Nov 25, 2004

DOCUMENT-IDENTIFIER: US 20040236945 A1
TITLE: Method and system for controlled media sharing in a network

Detail Description Paragraph:
[0100] In response to the client computer system 210 receiving the redirect command from web server 250, the media player application operating on client computer system 210 automatically transmits a new request and the time sensitive access key to content server 251 for delivery of one or more desired pieces of media content. The validity of the time sensitive access key is checked by content server 251. If invalid, unauthorized client computer 210 is redirected by content server 250 to protect against abuse of the system and unauthorized access to content server 251. If the time sensitive access key is valid, content server 251 retrieves the desired media content from content database 451 and delivers it to client computer system 210. It is noted that, in one embodiment, the delivered media content can be stored in hidden directories and/or custom file systems that may be hidden within client computer system 210 thereby preventing future unauthorized distribution. In one embodiment, an HTTP (hypertext transfer protocol) file delivery system is used to deliver the requested media files, meaning that the media files are delivered in their entirety to client computer system 210, as compared to streaming media which delivers small portions of the media file.

Detail Description Paragraph:
[0156] The present embodiment provides a mechanism for restricting recording of high fidelity media content delivered via one or more communication networks. The present embodiment delivers the high fidelity media content to registered clients while preventing unauthorized clients from directly receiving media content from a source database. Once the client computer system receives the media content, it can be stored in hidden directories and/or custom file systems that may be hidden to prevent subsequent unauthorized sharing with others. It is noted that various functionalities can be implemented to protect and monitor the delivered media content. For example, the physical address of the media content can be hidden from media content recipients. In another example, the directory address of the media content can be periodically changed. Additionally, an access key procedure and rate control restrictor can also be implemented to monitor and restrict suspicious media content requests. Furthermore, a copyright compliance mechanism, e.g., CCM 300, can be installed in the client computer system 210 to provide client side compliance with licensing agreements and copyright restrictions applicable to the media content. By implementing these and other functionalities, the present embodiment restricts access to and the distribution of delivered media content and provides a means for copyrighted media owner compensation.

Detail Description Paragraph:
[0183] For example, the present embodiment can cause the high fidelity media content to be stored in a volatile memory device, utilizing one or more hidden directories and/or custom file systems that may be hidden, where it may be cached for a limited period of time. Alternatively, the present embodiment can cause the high fidelity media content to be stored in a non-volatile memory device, e.g., 103 or data storage device 108. It is noted that the manner in which each of the delivered media content file(s) is stored, volatile or non-volatile, can be dependent upon the licensing restrictions and copyright agreements applicable to each media content file. It is further noted that in one embodiment, when a user of client computer system 210 turns the computer off or causes client computer system 210 to disconnect from the network, the media content stored in a volatile memory device is typically deleted therefrom.

Detail Description Paragraph:
[0184] Still referring to step 740, in another embodiment, the present embodiment can cause client computer system 210 to store the received media content in a non-volatile manner within a media application operating therein, or within one of its Internet browser applications

(e.g., Netscape Communicator.TM., Microsoft Internet Explorer.TM., Opera.TM., Mozilla.TM., and the like) so that delivered media content can be used in a repetitive manner. Further, the received media content can be stored in a manner making it difficult for a user to redistribute in an unauthorized manner, while allowing the user utilization of the received media content, e.g., by utilizing one or more hidden directories and/or custom file systems that may also be hidden. It is noted that by storing media content with client computer system 210 (when allowed by applicable licensing agreements and copyright restrictions), content server 251 does not need to redeliver the same media content to client computer system 210 each time its user desires to experience (e.g., listen to, watch, view, etc.) the media content file.

Detail Description Paragraph:
[0340] In the present embodiment, UCM 1800 further includes a secure player application 1810, a client communication application 1820, and a media storage container creator 1830. Media storage container creator 1830 is configured to allocate a portion of a memory unit coupled to the computer system in which UCM 1800 is installed, e.g., volatile memory 102 and/or non-volatile memory 103 of computer system 100 of FIG. 2. Media storage container creator 1830 utilizes the allocated portion of a memory unit and creates a protected media container file (e.g., a custom file system) into which received and/or availed instances of media, e.g., audio files, video files, multimedia files, documents, software, and the like, are stored. It is noted that in one embodiment, media content that is stored in a protected media contain file is, in addition to other encryptions applicable to the instance of media, encrypted local to the computer system on which the protected media file container is disposed. In an example, an instance of media 9090 is stored on both client computer system 1705 and source computer system 1715 of FIG. 17. Accordingly, media content 9090 is uniquely encrypted local to system 1705 and is uniquely encrypted local to system 1715. Additionally, in the context of the present invention, the term availed and/or availing refers to making available to a network, e.g., network 1700, an instance of media that may be stored in a protected media container file.

CLAIMS:

10. The method as recited in claim 1 further comprising: storing said instance of media content in a custom file system on a memory unit coupled to said source node, said custom file system accessible to a secure player application coupled to said source node.

14. The system as recited in claim 13 further comprising: a custom file system disposed on a memory unit coupled to said source node, said custom file system for storing said instance of media content.

32. The computer readable medium of claim 23 wherein said method further comprises storing said instance of media content in a custom file system on a memory unit coupled with said source node.

37. The method as recited in claim 33 further comprising: storing said instance of media content in an custom file system on a memory unit coupled to said node, wherein said instance of media content is stored in an encryption local to said node.

47. The system as recited in claim 43 wherein said node further comprises a custom file system disposed on a memory unit coupled thereto into which said instance of media content is stored.

56. The computer readable medium of claim 54 wherein said method further comprises storing said instance of media content in a custom file system on a memory unit coupled with said node.

94. The method as recited in claim 93 further comprising: transmitting the location of said instance of media content to said node, said instance of media content stored within an custom file system on a memory unit coupled to said source node.

109. The method as recited in claim 93 further comprising: storing said instance of media content in a custom file system on a memory unit coupled to said node and storing said instance of media content in a custom file system on a memory unit coupled to said source node.

119. The system as recited in claim 115 further comprising: a custom file system on a memory unit coupled to said source node for storing said instance of media content, said custom file system from which said secure player application avails to said plurality of nodes said

instance of media content.

124. The system as recited in claim 120 further comprising: a custom file system on a memory unit coupled to said node for storing said instance of media content, said custom file system from which said secure player application avails to said plurality of nodes said instance of media content.

143. The computer readable medium of claim 129 wherein said method further comprises storing said instance of media content in a custom file system on a memory unit coupled to said node and in a custom file system on a memory unit coupled to said node.

Previous Doc Next Doc Go to Doc#

First Hit Previous Doc Next Doc Go to Doc#

L9: Entry 4 of 14 File: PGPB Sep 9, 2004

DOCUMENT-IDENTIFIER: US 20040175000 A1
TITLE: Method and apparatus for a transaction-based secure storage file system

Detail Description Paragraph:
[0030] In one embodiment of the invention, the client device may include a pre-loaded shared library that can translate read/write/file name accesses into different read/write/file name accesses (without modifying the structure of the file system, and thus exposing the file system hierarchy). Alternatively, the shared library may also map read/write/file name accesses to a custom-implemented file system. The mapping may take place in the library itself, or in a process with which the library communicates (e.g., via shared memory (SHM)) and which acts on behalf of the library. The custom file system may reside on top of the existing file system and be realized as a set of opaque files, or alternatively, the file system may include of access to a raw block device (i.e., a floppy disk, tape drive, etc.).

Previous Doc Next Doc Go to Doc#

L9: Entry 5 of 14 File: PGPB Jun 24, 2004

DOCUMENT-IDENTIFIER: US 20040123103 A1
TITLE: Method for redirecting of kernel data path for controlling recording of media

Detail Description Paragraph:
[0081] In response to the client computer system 210 receiving the redirect command from web server 250, the media player application operating on client computer system 210 automatically transmits a new request and the time sensitive access key to content server 251 for delivery of one or more desired pieces of media content. The validity of the time sensitive access key is checked by content server 251. If invalid, unauthorized client computer 210 is redirected by content server 250 to protect against abuse of the system and unauthorized access to content server 251. If the time sensitive access key is valid, content server 251 retrieves the desired media content from content database 451 and delivers it to client computer system 210. It is noted that, in one embodiment, the delivered media content can be stored in hidden directories and/or custom file systems that may be hidden within client computer system 210 thereby preventing future unauthorized distribution. In one embodiment, an HTTP (hypertext transfer protocol) file delivery system is used to deliver the requested media files, meaning that the media files are delivered in their entirety to client computer system 210, as compared to streaming media which delivers small portions of the media file.

Detail Description Paragraph:
[0141] The present embodiment provides a mechanism for restricting recording of high fidelity media content delivered via one or more communication networks. The present embodiment delivers the high fidelity media content to registered clients while preventing unauthorized clients from directly receiving media content from a source database. Once the client computer system receives the media content, it can be stored in hidden directories and/or custom file systems that may be hidden to prevent subsequent unauthorized sharing with others. It is noted that various functionalities can be implemented to protect and monitor the delivered media content. For example, the physical address of the media content can be hidden from media content recipients. In another example, the directory address of the media content can be periodically changed. Additionally, an access key procedure and rate control restrictor can also be implemented to monitor and restrict suspicious media content requests. Furthermore, a copyright compliance mechanism, e.g., CCM 300, can be installed in the client computer system 210 to provide client side compliance with licensing agreements and copyright restrictions applicable to the media content. By implementing these and other functionalities, the present embodiment restricts access to and the distribution of delivered media content and provides a means for copyrighted media owner compensation.

Detail Description Paragraph:
[0167] In step 740 of FIG. 7C, upon receiving the requested high fidelity media content from content server 251, the present embodiment causes client computer system 210 to store the delivered media content in a manner that is ready for presentation, e.g., play. The media content is stored in client computer system 210 in a manner that restricts unauthorized redistribution. For example, the present embodiment can cause the high fidelity media content to be stored in a volatile memory device, utilizing one or more hidden directories and/or custom file systems that may be hidden, where it may be cached for a limited period of time. Alternatively, the present embodiment can cause the high fidelity media content to be stored in a non-volatile memory device, e.g., 103 or data storage device 108. It is noted that the manner in which each of the delivered media content file(s) is stored, volatile or non-volatile, can be dependent upon the licensing restrictions and copyright agreements applicable to each media content file. It is further noted that in one embodiment, when a user of client computer system 210 turns the computer off or causes client computer system 210 to disconnect from the network, the media content stored in a volatile memory device is typically deleted therefrom.

Detail Description Paragraph:
[0168] Still referring to step 740, in another embodiment, the present embodiment can cause client computer system 210 to store the received media content in a non-volatile manner within a media application operating therein, or within one of its Internet browser applications (e.g., Netscape Communicator.TM., Microsoft Internet Explorer.TM., Opera.TM., Mozilla.TM., and the like) so that delivered media content can be used in a repetitive manner. Further, the received media content can be stored in a manner making it difficult for a user to redistribute in an unauthorized manner, while allowing the user utilization of the received media content, e.g., by utilizing one or more hidden directories and/or custom file systems that may also be hidden. It is noted that by storing media content with client computer system 210 (when allowed by applicable licensing agreements and copyright restrictions), content server 251 does not need to redeliver the same media content to client computer system 210 each time its user desires to experience (e.g., listen to, watch, view, etc.) the media content file.

Previous Doc Next Doc Go to Doc#

L9: Entry 6 of 14 File: PGPB May 27, 2004

DOCUMENT-IDENTIFIER: US 20040103300 A1
TITLE: Method of controlling recording of media

Detail Description Paragraph:
[0080] In response to the client computer system 210 receiving the redirect command from web server 250, the media player application operating on client computer system 210 automatically transmits a new request and the time sensitive access key to content server 251 for delivery of one or more desired pieces of media content. The validity of the time sensitive access key is checked by content server 251. If invalid, unauthorized client computer 210 is redirected by content server 250 to protect against abuse of the system and unauthorized access to content server 251. If the time sensitive access key is valid, content server 251 retrieves the desired media content from content database 451 and delivers it to client computer system 210. It is noted that, in one embodiment, the delivered media content can be stored in hidden directories and/or custom file systems that may be hidden within client computer system 210 thereby preventing future unauthorized distribution. In one embodiment, an HTTP (hypertext transfer protocol) file delivery system is used to deliver the requested media files, meaning that the media files are delivered in their entirety to client computer system 210, as compared to streaming media which delivers small portions of the media file.

Detail Description Paragraph:
[0128] The present embodiment provides a mechanism for restricting recording of high fidelity media content delivered via one or more communication networks. The present embodiment delivers the high fidelity media content to registered clients while preventing unauthorized clients from directly receiving media content from a source database. Once the client computer system receives the media content, it can be stored in hidden directories and/or custom file systems that may be hidden to prevent subsequent unauthorized sharing with others. It is noted that various functionalities can be implemented to protect and monitor the delivered media content. For example, the physical address of the media content can be hidden from media content recipients. In another example, the directory address of the media content can be periodically changed. Additionally, an access key procedure and rate control restrictor can also be implemented to monitor and restrict suspicious media content requests. Furthermore, a copyright compliance mechanism, e.g., CCM 300, can be installed in the client computer system 210 to provide client side compliance with licensing agreements and copyright restrictions applicable to the media content. By implementing these and other functionalities, the present embodiment restricts access to and the distribution of delivered media content and provides a means for copyrighted media owner compensation.

Detail Description Paragraph:
[0154] In step 740 of FIG. 7C, upon receiving the requested high fidelity media content from content server 251, the present embodiment causes client computer system 210 to store the delivered media content in a manner that is ready for presentation, e.g., play. The media content is stored in client computer system 210 in a manner that restricts unauthorized redistribution. For example, the present embodiment can cause the high fidelity media content to be stored in a volatile memory device, utilizing one or more hidden directories and/or custom file systems that may be hidden, where it may be cached for a limited period of time. Alternatively, the present embodiment can cause the high fidelity media content to be stored in a non-volatile memory device, e.g., 103 or data storage device 108. It is noted that the manner in which each of the delivered media content file(s) is stored, volatile or non-volatile, can be dependent upon the licensing restrictions and copyright agreements applicable to each media content file. It is further noted that in one embodiment, when a user of client computer system 210 turns the computer off or causes client computer system 210 to disconnect from the network, the media content stored in a volatile memory device is typically deleted therefrom.

Detail Description Paragraph:
[0155] Still referring to step 740, in another embodiment, the present embodiment can cause client computer system 210 to store the received media content in a non-volatile manner within a media application operating therein, or within one of its Internet browser applications (e.g., Netscape Communicator.TM., Microsoft Internet Explorer.TM., Opera.TM., Mozilla.TM., and the like) so that delivered media content can be used in a repetitive manner. Further, the received media content can be stored in a manner making it difficult for a user to redistribute in an unauthorized manner, while allowing the user utilization of the received media content, e.g., by utilizing one or more hidden directories and/or custom file systems that may also be hidden. It is noted that by storing media content with client computer system 210 (when allowed by applicable licensing agreements and copyright restrictions), content server 251 does not need to redeliver the same media content to client computer system 210 each time its user desires to experience (e.g., listen to, watch, view, etc.) the media content file.

Previous Doc Next Doc Go to Doc#

L9: Entry 9 of 14 File: PGPB May 30, 2002

DOCUMENT-IDENTIFIER: US 20020065840 A1
TITLE: Method and system for creating and managing common and custom storage devices in a computer network

Detail Description Paragraph:
[0044] Once the primary or an alternate root storage device has been activated, the booting method mounts the file systems from the activated common root storage device (box 768). The present invention then determines whether the level of the common root storage device (primary or alternate) matches the custom root storage device level (box 772). If the two levels do not match, the operating system files of the custom root storage device are updated based on the common root storage device level (box 776). Otherwise, the booting process is completed (box 780).

CLAIMS:

15. The method of claim 12, further comprising determining whether a level of the activated common root storage device level matches a level of the custom root storage device and, if not, then updating the operating system files on the custom root storage device based on the common root storage device level.

L14: Entry 6 of 8 File: USPT Nov 20, 2001

DOCUMENT-IDENTIFIER: US 6321219 B1
TITLE: Dynamic symbolic links for computer file systems

Abstract Text (1):
A user creates a rule defining file objects to appear at an arbitrary insertion point in a hierarchical file structure. Every request for a file, directory, etc. at the insertion point invokes the rule and constructs a set of dynamic links to actual locations of all file objects that satisfy the rule. Any operations performed by the program on the objects take place transparently on the objects at their actual locations. Between requests, the actual links go away, and only the rule for constructing them remains. An indexer operates continually to maintain current information on all files in the structure.

Brief Summary Text (2):
The present invention relates to electronic data processing, and more specifically concerns the creation and use of symbolic links for organizing file structures in a computer.

Brief Summary Text (3):
Most computer operating systems provide facilities for storing individual files in a structured arrangement from which they can be accessed by user application programs. Hierarchical file systems, the most common type, posit a root directory for each logical or physical storage device such as a disk drive. The root directory can contain individual files, and can also contain subdirectories which in turn contain files and subdirectories to any desired level. Directories and subdirectories are identical in function, and are sometimes called folders or other names. Some file systems, the best known of which is UNIX, attach file systems from different devices into a single file hierarchy. This is referred to as "mounting" a file system.

Brief Summary Text (4):
Users employ directory structures to organize their data and programs. For example, a storage device designated "C:" may contain legal documents in a directory "C:.backslash.LegalDocs". A user wishes to organize the documents by docket number, and accordingly sets up a subdirectory for each one: "C:.backslash.LegalDocs.backslash.111803", "C:.backslash.LegalDocs.backslash.98007", and so on. Each directory at the lowest level then contains files dealing with that particular docket. However, another user may desire to organize the same files on the same storage device by author, using directories such as "C:.backslash.LegalDocs.backslash.Norm_D_Plume", "C:.backslash.LegalDocs.backslash.Sue_Donym", etc. A third user may desire all files created within the current month to be in a single directory "C:.backslash.LegalDocs.backslash.Recent". But, if the operating system can only store each file in a single folder, then the files can be organized in only one way. Application programs such as document-control utilities sidestep this problem by allowing users to create profiles for each file, and then accessing the files in response to users' queries for files having certain characteristics in the profiles. Although these programs function well, they function with only one application program, or with applications that adhere to certain protocols or standards. Placing a number of existing files within a document-control system requires manually generating profiles for each file. These programs tend to be large, expensive, and difficult to configure or modify. In addition, switching from one document-control system to a different one usually requires redoing the profiles of all the files. Also, commercial document-control systems are overkill in many small tasks or ad-hoc situations.

Brief Summary Text (5):
Another approach introduces the concept of symbolic links. Many operating systems include a facility that allows a user or an administrator to create a link between an existing file or directory and a new name. Thereafter, both the new name and the old name refer to the same file

or directory. Changes made during an access under either name appear under a later access under the other name as well. These links provide aliases for files, different ways to access the same physical entity.

Brief Summary Text (6):
Conventional links, however, are "static" symbolic links. They require explicit create and delete actions on behalf of a computer user. The links must be manually removed when no longer needed, even after the physical files or directories to which they refer have been removed from the system. The administration of static symbolic links quickly becomes unwieldy as the number of links increases. Because of oversights or interruptions, some files that should be linked will not be linked, and broken links will remain after their underlying files have been removed or renamed. Although system scripts or programs can be written by systems administrators to automate portions of link-management tasks, they are error-prone and have limited function. Such software must be run periodically, and links are likely to become obsolete between runs. Pushed beyond relatively simple file structures, static links obscure the relationships between file-system objects. Further, the creation and maintenance of static links are difficult enough to deter naive or casual users from even attempting to learn how they work.

Brief Summary Text (7):
Therefore, the file structures of many operating systems lack an effective facility for handling multiple organizations of files, folders, directories, and other objects in a manner that is error-free, transparent to all application programs, and simple to learn.

Brief Summary Text (10):
A utility program accessible to users receives definitions of rules or associations for creating symbolic links among particular file-system objects such as files and directories. Rule creation is simple and direct, and the rules can be general and powerful. A file-system component called a dynamic link driver detects operations occurring at points in the file-system name space where rules have been defined, creates symbolic links among objects as specified by the rules. This link driver can be inserted in the file-management or other modules of conventional operating systems without extensive modifications, perhaps even as a third-party device driver. Because the link driver creates and handles the links within the file system itself, the symbolic links are transparent to all application programs that access files and directories, and even to other levels of the file system itself. That is, an application accesses a linked file or directory with exactly the same mechanisms with which it accesses any file or directory; with no change whatsoever to the application code, and without complying with any additional standard or protocol. A rules component or query processor, either within the file-system link driver or located separately, stores and accesses the data required to carry out the rules. The rules defining a given set of dynamic symbolic links can be satisfied directly by the link driver; that is, the link driver can directly execute and satisfy the rule. Alternatively, the rule can be forwarded to any supported query processor. For example, Microsoft.RTM. WindowsNT.RTM. 5 supports full content indexing on all files, and its Windows content-indexing component can be the target of the dynamic symbolic link rule. The content-indexing component can receive the query, execute it, and return the result to the link driver, which can use the result to execute the file or directory operation.

Brief Summary Text (11):
The utility program receives from a user a request to associate certain file-system objects. The utility requests the user to specify a location in the file system name space for the association, and to specify a rule, association, relation, or query (these terms are interchangeable) for the insertion point. The utility then installs that rule at the insertion point. A subsequent request from a user application program--or, equivalently, from the system browser--for a file-system object such as a file or a directory at the insertion point invokes the file-system component to call the rules component to determine which objects at what actual locations fit the rule for that insertion point. The file system creates the appropriate links for objects that satisfy the rule, effectively plugging their names or other identifiers into the file system at the insertion point. The file system thereafter returns the objects to the application program transparently, exactly as though they actually existed at the insertion point. The next time the application--or any other application--requests the object, the rules are executed again, and new links are created; that is, the links themselves do not exist between accesses, but only the rules for creating them. Therefore, should a user delete a linked object or move it to a different location during an access, the object might no longer exist or satisfy the rule, and a subsequent access to the same insertion point will simply not

create a link to that object. The links thus require no maintenance or administration whatsoever, and there is no time interval during which they are incorrect or obsolete.

Detailed Description Text (4):
This section provides a brief, general description of a suitable computing environment in which the invention may be implemented. The invention will hereinafter be described in the general context of computer-executable program modules containing instructions executed by a personal computer (PC). Program modules include routines, programs, objects, components, data structures, etc. that perform particular tasks or implement particular abstract data types. Those in the art will appreciate that the invention may be practiced with other computer-system configurations, including handheld devices, multiprocessor systems, microprocessor-based programmable consumer electronics, network PCs, minicomputers, mainframe computers, and the like. The invention may also be practiced in distributed computing environments where tasks are performed by remote processing devices linked through a communications network. In a distributed computing environment, program modules may be located in both local and remote memory storage devices.

Detailed Description Text (5):
FIG. 1 employs a general-purpose computing device in the form of a conventional personal computer 20, which includes processing unit 21, system memory 22, and system bus 23 that couples the system memory and other system components to processing unit 21. System bus 23 may be any of several types, including a memory bus or memory controller, a peripheral bus, and a local bus, and may use any of a variety of bus structures. System memory 22 includes read-only memory (ROM) 24 and random-access memory (RAM) 25. A basic input/output system (BIOS) 26, stored in ROM 24, contains the basic routines that transfer information between components of personal computer 20. BIOS 24 also contains start-up routines for the system. Personal computer 20 further includes hard disk drive 27 for reading from and writing to a hard disk (not shown), magnetic disk drive 28 for reading from and writing to a removable magnetic disk 29, and optical disk drive 30 for reading from and writing to a removable optical disk 31 such as a CD-ROM or other optical medium. Hard disk drive 27, magnetic disk drive 28, and optical disk drive 30 are connected to system bus 23 by a hard-disk drive interface 32, a magnetic-disk drive interface 33, and an optical-drive interface 34, respectively. The drives and their associated computer-readable media provide nonvolatile storage of computer-readable instructions, data structures, program modules and other data for personal computer 20. Although the exemplary environment described herein employs a hard disk, a removable magnetic disk 29 and a removable optical disk 31, those skilled in the art will appreciate that other types of computer-readable media which can store data accessible by a computer may also be used in the exemplary operating environment. Such media may include magnetic cassettes, flash-memory cards, digital versatile disks, Bernoulli cartridges, RAMs, ROMs, and the like.

Detailed Description Text (6):
Program modules may be stored on the hard disk, magnetic disk 29, optical disk 31, ROM 24 and RAM 25. Program modules may include operating system 35, one or more application programs 36, other program modules 37, and program data 38. A user may enter commands and information into personal computer 20 through input devices such as a keyboard 40 and a pointing device 42. Other input devices (not shown) may include a microphone, joystick, game pad, satellite dish, scanner, or the like. These and other input devices are often connected to the processing unit 21 through a serial-port interface 46 coupled to system bus 23; but they may be connected through other interfaces not shown in FIG. 1, such as a parallel port, a game port, or a universal serial bus (USB). A monitor 47 or other display device also connects to system bus 23 via an interface such as a video adapter 48. In addition to the monitor, personal computers typically include other peripheral output devices (not shown) such as speakers and printers.

Detailed Description Text (10):
FIG. 2 shows a specific example of a multi-level hierarchical file structure 200 of the type implemented by many conventional computer operating systems, such as Microsoft Windows NT. File structure 200 is organized as a tree having a root node 201 containing a root directory. The root directory contains some files 203, and two subdirectories 204 and 205. The terms `directory` and `subdirectory` are equivalent for the present purpose. The term `file object` is used herein to denote either a directory or a file, and can also be applied to any other resource that can be stored in or referenced by a file system, such as a named pipe or a mailslot. Directory 204 contains two further subdirectories 207 and 208, and therefore defines another node 209. Directory 210 at node 211 contains files and subdirectory 213 at node 214.

Directory 213 contains files 215 and yet another directory 216. The dotted lines at node 217 indicate further files and subdirectories to any desired level. Directory 218, located at node 209, may also contain further objects at node 219. Directory 205 is shown as containing files 220 at its node 221, and also contains an insertion point 230. An insertion point is a type of file object for attaching a rule according to the invention.

Detailed Description Text (12):
The reparse information for effecting the invention includes a representation of a rule or query 231 that defines certain characteristics of the files or other objects for which dynamic links are to be constructed. These files, directories, etc. are actually located in other directories scattered arbitrarily throughout tree 200. For example one or more of the files 212, 216, and the directory 218, located in directories 210, 213, and 204 might satisfy the conditions of rule 231. Dynamic links, shown as lines 232-234, then cause such files to appear to reside in a directory located at insertion point 230. Continuing the previous example, if the name of root directory is "C:.backslash.y ", the name of directory 205 is "LegalDocs", and the name of the faux directory at point 230 is "Sue_Donym", then a request to list the contents of directory "C:.backslash.LegalDocs.backslash.Sue_Donym" returns the names, attributes, and other information for a set of phantom files and directories 235 that are actually located at other places throughout the file structure 200. The requesting program, however, can deal with them as though they were in fact located at the insertion point 230. For example, a user application program can open, modify, move, or delete a file. Any action upon such a file or directory object takes place upon the requested object at its actual location, transparently to the application. When the requesting program closes its access, links 232-234 go away.

Detailed Description Text (13):
FIG. 3 shows a system 300 for creating and manipulating dynamic links such as 232-234, FIG. 2. Almost all operating systems such as 35, FIG. 1, include a component known as a file system, file processor, file manager, or similar component. In the example WindowsNT operating system, input/output manager (NTIO) 310 fulfills this overall function as well as other functions not relevant to the present invention; therefore, block 310 is referred to herein as the file system or file manager. (By way of background, the WindowsNT 5.0 NTIO component of this embodiment controls all storage-related I/O and provides the file system interface, but it calls file-system drivers to handle the actual file operations. That is, NTIO does not handle I/O in and of itself. Instead, it calls the appropriate client file-system drivers, called NTFS in WindowsNT.) OS components such as manager 310 typically run in a kernel mode that protects them from user-level programs. A user-level association utility 320 defines which file objects will be linked to which insertion points 230, FIG. 2. The definition can be provided by a user employing input/output (110) devices such as keyboard 40 and display 47; alternatively, some other program can provide or modify the definition. Any application program 330 may request a particular file service in an entirely conventional manner by issuing a standard API (application program interface) to I/O manager 310. Manager 310 processes the API and ultimately issues a request to a storage device 330 that contains data for satisfying the request. The 110 manager processes this data, and either returns it to requesting program 310 or uses it to produce some other reply. In some cases, such as an incorrect syntax in the request, manager 310 turns the request around itself, producing a reply without ever reaching a storage device.

Detailed Description Text (14):
Like most modern file systems, manager 310 has a layered internal structure and employs pluggable components for adding and modifying functionality. Conventional device drivers 311, for example, translate requests in a common internal format into commands and data understood by specific kinds of storage devices, and reformat return data from the devices back into the common format. File system drivers 312 and similar modules perform intermediate functions such as error checking and file sharing. Filter drivers 313 form a newer class of file-system component. They detect a designated condition, flag, or characteristic at a specific I/O-manager level, and intercept a request or returned data. Filter drivers are drivers that receive all I/O requests sent to the target file system and at each level have an opportunity to satisfy the request, fail the request, modify the request, or issue additional requests to satisfy the request that they have received. A filter driver can intercept a file operation between any two layers traveling from an application program downward through the file manager to a physical storage device, and again on the way back from the device to the program. The filter driver then performs a function and can place the request or data back into the manager. Filter drivers are transparent to the operation of I/O manager 310. Conventional filter drivers

perform functions such as virus detection, data encryption, and hierarchical storage management. Filter drivers can invoke the services of other modules in the file system, other components of the operating system, and any other program at any level.

Detailed Description Text (15):
System 300 couples a dynamic-link (DL) filter driver 313 to I/O manager 310 for processing dynamic links during application-program accesses to the file system. Driver 313 also connects to a kernel-mode client 341 of a database management subsystem (DBMS) 340. Clients of this type provide a local user interface to a larger DBMS server located either in the same computer or at a remote facility. In this example, DBMS server 342 operates in the user space of the local computer 20, FIG. 1, and connects to client 341 via a conventional block of shared memory 343. A client for use in the invention entails a relatively small and easily written code module for invoking database queries in response to inputs from DL driver 313. Placing the client in the kernel space allows more direct communication with driver 313.

Detailed Description Text (17):
File indexer 345 is a facility found in the Enterprise Edition of WindowsNT. Its normal function is to continually update a relational database 346 holding certain characteristics of all the objects stored in file structure 200. That is, columns of database 346 can hold the type, size, date, and other attributes of files and directories. Database 346 can also employ a database to index file contents: words in a document, names in a spreadsheet, and so forth. In system 300, search engine 344 parses a query at an insertion point, and then DBMS server 342 redirects the query to database 346 of file indexer 345.

Detailed Description Text (19):
FIG. 4 is a flowchart of a method 400 for implementing association utility 320, FIG. 3. A user (or another program) requests at block 410 that dynamic symbolic links be defined. Block 420 receives the user's designation of a point such as 230 in the directory structure 200, FIG. 2, for insertion of the links. Block 430 requests the services of I/O manager 310 to create a reparse point at the designated insertion point. The illustrative WindowsNT file manager creates an empty directory with its reparse attribute set. Block 440 receives the user's definition of a rule for specifying which files should appear at the insertion point. In this implementation, the rule has the form of a query in the wellknown SQL query language. For example, the query

Detailed Description Text (24):
FIG. 5 diagrams a method 500 for creating and manipulating dynamic symbolic links that have been defined by method 400, FIG. 4.

Detailed Description Text (25):
Block 510 initiates method 500 in file system 310, whenever it receives a request for a file-system service or operation form a user-level application program or from some other source. This description uses a request for a directory listing as an illustrative example of a file operation. Block 520 traverses file structure 200 searching for the requested file object. If block 521 does not encounter a reparse point, block 530 in the file system obtains the object, using the appropriate file-system drivers and device drivers shown in FIG. 3. Block 531 returns the object to the requesting application. For a directory request, the actual form of the object is a list of names, attributes, locations, and other information for each file in the directory. Where the directory is a conventional one, the file information is that of the actual files in the entries for that directory.

Detailed Description Text (26):
If block 521 encounters a reparse point having a return code indicating that it is an insertion point for dynamic links, then DL driver 313 intercepts the STATUS_REPARSE error code in block 522. (Without this intercept, any error code would be passed all the way back to the requester.) Block 540 sends the query text from the reparse point to client 341 of DBMS subsystem 340, which connects DBMS server 342 and passes the query to it for parsing and execution in a conventional manner. Filter driver 313 matches the "remainingName" against the list of files it gets from the DBMS, and constructs the new name of the file to be opened. Although server 342 could execute the query on a database that it manages, block 541 instead redirects the query to file database 346. Block 550 indicates the action of indexer 345 in continually indexing all the files in structure 200 asynchronously to the remainder of process 500, so as to maintain file database 346 current at all times. Method 500 can, however, employ

other mechanisms for providing a current database or some other form of file information for creating dynamic links. For example, the file-structure directories themselves can form a searchable database. More generally, the invention can use any form of query processor, and can alternatively employ rule processors of other kinds, such as those found in expert systems.

Detailed Description Text (27):
Blocks 560 return database information for those files in structure 200 that satisfy the query. Block 561 opens a conventional cursor on the database. For each database row that satisfies the query, block 562 increments the cursor to point to that row in turn as the current row. Block 563 fetches the current row. Block 564 reformats the information in the row into proper form if required, and block 565 stores the information in the file object. For a directory-listing request, each row represents a file, and the columns contain file information, as described above. One of the file-information items is the actual location of the file in file structure 200. This location thus is a link to the file. When all rows have been processed, block 562 causes block 530 in I/O manager 310 to get the file object. Block 531 then returns the object to the requesting program 330 in the normal manner. In many file systems, block 531 returns a handle by which the requesting program can access a file object, rather than returning the object itself. Some operations, such as the directory listing used herein as an example, then employ the handle to obtain and return one file at a time through blocks 530 and 531 as the requesting program issues successive APIs for them, rather than returning all the information at once.

Detailed Description Text (28):
In this way, conventional application programs request and receive file objects in an entirely transparent manner, without any modification whatsoever. For example, an application program requests a directory listing using a conventional API for that purpose. The file system returns it via another standard API. The format of the listing is conventional, and the application can then open one of the files in the listing in its usual way. Method 500 at this point reads the file location at block 520, which now points to the actual location of the file in structure 200, rather than to the phantom directory at insertion point 230. File-system block 530 retrieves the file from its actual location, and block 531 returns it to the requesting application, all in a conventional manner. The only difference is that the file location information now forms a dynamic link to the actual location, rather than to the phantom directory at the insertion point that the user program thought it had requested. Any operation that application program 330 performs on the file, including modification, renaming, or deletion, is carried out on the real file at its actual location. Whenever the application releases an access via a standard file-system API, its file object disappears as it normally would. Therefore, the dynamic link merely disappears. The next request, by that application or by any other program, to insertion point 230 reexecutes method 500. Because block 550 continually updates the file-structure database, the links dynamically constructed by blocks 560 from their defining query might well differ from the links built during the previous request.

Detailed Description Text (29):
FIG. 6 is a flowchart 600 showing how method 500 retrieves dynamically linked files for an application program. Block 610 receives a request to, for example, open a file at block 610. Block 610 functions in exactly the same way as block 510, whether or not the requested file has a dynamic link. Block 620 follows the path of the directory returned by block 531, in exactly the same manner as block 530. The only difference is that the path to the actual location of the file was not placed in the directory by the file manager itself, but rather by DL filter driver 313. Because the file location actually exists in the file structure, it is treated like any other file, directory, or even another insertion point. Block 630 gets the file from its actual location just as block 530 would, and block 631 returns the file to the requesting program in the same way as block 531, FIG. 5. All of these blocks execute in the file manager 310 in a manner entirely transparent to the application program. That is, the application program need have no knowledge of the dynamic link, and need not request or receive the file any differently than a normal file.

Detailed Description Text (30):
Although dynamic symbolic links as described above are transparent to conventional application programs in almost all respects, there are a few differences. For example, the existence of multiple files having the same name is possible when the files are located in different actual directories. If such identically named files satisfy a query, then a request by an application

file via an insertion point for a file having that name will produce an ambiguity. This issue could be resolved in any of several ways, such as by appending unique identifiers to the name of the different actual files. Directories at insertion points are read-only, so that no new files can be created in these directories; again, this facility can be provided if desired.

Detailed Description Text (31):
Many variations and alternatives within the scope of the following claims will appear to those skilled in the art. In particular, although the above description locates much of the invention in the kernel of an operating system, any component of the invention can be located in any defined level or space in the programming environment of a computer. Also, the terms "operating system," "file manager," "indexer," and so forth must be construed broadly. Software or hardware components having different names and overall functions may also serve the present invention. It must also be remembered that the methods defined below can be performed in any temporal order, except where specifically indicated in the claims.

CLAIMS:

1. A method executed on a programmable computer for managing a file structure of file objects in a file system, the method comprising:

defining a rule specifying a desired characteristic for a set of the file objects in the file system;

designating a point in the file structure as an insertion point;

persistently associating the rule with the insertion point;

receiving a file-system request directed to the insertion point;

constructing a set of dynamic links to the set of the file objects that satisfy the rule; and

returning the dynamic links in a response to the request.

3. A method according to claim 1 wherein the file structure is a multi-level hierarchical structure.

7. A method according to claim 1 wherein the receiving step comprises:

receiving in the file system a name indicating one of a number of points in the file structure;

detecting that the one point is the insertion point.

8. A method according to claim 1 wherein the constructing step comprises:

searching the file structure for a set of file objects that satisfy the rule.

9. A method according to claim 8 wherein the file objects in the structure have respective locations within the structure, and wherein the constructing step further comprises

returning the actual location of each of the files in the set as a dynamic link for that file.

11. A method according to claim 1 further comprising indexing the file objects in the file structure asynchronously with respect to the remaining steps.

12. A method according to claim 1 wherein the indexing step includes building a database of the file objects in the file structure.

14. A programmable digital computer for executing application programs that access file objects located at multiple points in a file structure of a file system, the computer comprising:

storage means for storing the file objects;

input-output means for receiving a rule and for designating one of said multiple points as an insertion point, wherein the rule is persistently associated with the insertion point;

query processor means for determining which of the file objects at any of the multiple points satisfy the rule;

file processor means for managing the file structure;

dynamic link driver means coupled to the file processor means and to the query processor for detecting an access request directed to the insertion point, and in response thereto for constructing a set of dynamic links from the insertion point to those of the file objects that satisfy the rule.

21. A method executed on a computer for managing a file structure of file objects at multiple different locations, comprising:

receiving a request for a file object at one of the locations;

detecting that the one location is an insertion point;

accessing a rule persistently associated with the insertion point;

determining the locations of the file structure of a set of file objects that satisfy the rule;

constructing a set of dynamic links from the insertion point to each of the locations of the file objects in the set;

returning the dynamic links as a response to the request.

23. A method according to claim 21 wherein the file structure is a multi-level hierarchical structure.

25. A method according to claim 24 wherein determining the locations in the file structure of a set of file objects that satisfy the rule comprises searching a database for file objects that satisfy the query.

27. A method according to claim 25 further comprising indexing the file structure so as to construct the database.

29. A method according to claim 21 wherein the rule designates certain characteristics of the set of file objects, and wherein constructing the dynamic links comprises:

searching the file structure for the set of the file objects having the certain characteristics;

constructing a separate dynamic link for each of the file objects in the set to the insertion point.

32. A method according to claim 31 wherein accessing the one file object comprises:

receiving a request for the one file object directed to the insertion point;

reading the actual location of the one file object in the file structure from the dynamic link for the one file object;

returning the one file object from its actual location.

33. A programmable digital computer for executing application programs that access file objects located at multiple points in a file structure of a file system, the computer comprising:

a file manager for receiving file-object requests from the application program and for returning requested file objects thereto;

a link driver coupled to the file manager for intercepting requests directed to a predefined insertion point in the file structure and for fetching a rule persistently associated with the insertion point;

a subsystem coupled to the link driver for determining a set of the file objects at any of the multiple points in the file structure that satisfy the rule,

the link driver further returning the set of file objects as the requested file objects.

37. A computer according to claim 33 further including an indexer for constructing and continually updating a database of the file objects in the file structure.

39. A method executed on a computer for creating dynamic symbolic links in a file structure of file objects at multiple different locations in a file system, comprising:

receiving a designation of a desired location for the set of file objects;

creating an insertion point at the desired location;

receiving a rule specifying a desired characteristic of the file objects in the set, wherein each file object in the set satisfies the rule;

receiving a rule specifying the file objects in the set;

persistently associating the rule with the insertion point.

44. A method according to claim 39 wherein the rule is stored in the file structure at the insertion point.

Previous Doc Next Doc Go to Doc#

L14: Entry 6 of 8 File: USPT Nov 20, 2001

DOCUMENT-IDENTIFIER: US 6321219 B1
TITLE: Dynamic symbolic links for computer file systems

Abstract Text (1):
A user creates a rule defining file objects to appear at an arbitrary insertion point in a hierarchical file structure. Every request for a file, directory, etc. at the insertion point invokes the rule and constructs a set of dynamic links to actual locations of all file objects that satisfy the rule. Any operations performed by the program on the objects take place transparently on the objects at their actual locations. Between requests, the actual links go away, and only the rule for constructing them remains. An indexer operates continually to maintain current information on all files in the structure.

Brief Summary Text (2):
The present invention relates to electronic data processing, and more specifically concerns the creation and use of symbolic links for organizing file structures in a computer.

Brief Summary Text (3):
Most computer operating systems provide facilities for storing individual files in a structured arrangement from which they can be accessed by user application programs. Hierarchical file systems, the most common type, posit a root directory for each logical or physical storage device such as a disk drive. The root directory can contain individual files, and can also contain subdirectories which in turn contain files and subdirectories to any desired level. Directories and subdirectories are identical in function, and are sometimes called folders or other names. Some file systems, the best known of which is UNIX, attach file systems from different devices into a single file hierarchy. This is referred to as "mounting" a file system.

Brief Summary Text (4):
Users employ directory structures to organize their data and programs. For example, a storage device designated "C:" may contain legal documents in a directory "C:.backslash.LegalDocs". A user wishes to organize the documents by docket number, and accordingly sets up a subdirectory for each one: "C:.backslash.LegalDocs.backslash.111803", "C:.backslash.LegalDocs.backslash.98007", and so on. Each directory at the lowest level then contains files dealing with that particular docket. However, another user may desire to organize the same files on the same storage device by author, using directories such as "C:.backslash.LegalDocs.backslash.Norm_D_Plume", "C:.backslash.LegalDocs.backslash.Sue_Donym", etc. A third user may desire all files created within the current month to be in a single directory "C:.backslash.LegalDocs.backslash.Recent". But, if the operating system can only store each file in a single folder, then the files can be organized in only one way. Application programs such as document-control utilities sidestep this problem by allowing users to create profiles for each file, and then accessing the files in response to users' queries for files having certain characteristics in the profiles. Although these programs function well, they function with only one application program, or with applications that adhere to certain protocols or standards. Placing a number of existing files within a document-control system requires manually generating profiles for each file. These programs tend to be large, expensive, and difficult to configure or modify. In addition, switching from one document-control system to a different one usually requires redoing the profiles of all the files. Also, commercial document-control systems are overkill in many small tasks or ad-hoc situations.

Brief Summary Text (5):
Another approach introduces the concept of symbolic links. Many operating systems include a facility that allows a user or an administrator to create a link between an existing file or directory and a new name. Thereafter, both the new name and the old name refer to the same file

or directory. Changes made during an access under either name appear under a later access under the other name as well. These links provide aliases for files, different ways to access the same physical entity.

Brief Summary Text (6):
Conventional links, however, are "static" symbolic links. They require explicit create and delete actions on behalf of a computer user. The links must be manually removed when no longer needed, even after the physical files or directories to which they refer have been removed from the system. The administration of static symbolic links quickly becomes unwieldy as the number of links increases. Because of oversights or interruptions, some files that should be linked will not be linked, and broken links will remain after their underlying files have been removed or renamed. Although system scripts or programs can be written by systems administrators to automate portions of link-management tasks, they are error-prone and have limited function. Such software must be run periodically, and links are likely to become obsolete between runs. Pushed beyond relatively simple file structures, static links obscure the relationships between file-system objects. Further, the creation and maintenance of static links are difficult enough to deter naive or casual users from even attempting to learn how they work.

Brief Summary Text (7):
Therefore, the file structures of many operating systems lack an effective facility for handling multiple organizations of files, folders, directories, and other objects in a manner that is error-free, transparent to all application programs, and simple to learn.

Brief Summary Text (10):
A utility program accessible to users receives definitions of rules or associations for creating symbolic links among particular file-system objects such as files and directories. Rule creation is simple and direct, and the rules can be general and powerful. A file-system component called a dynamic link driver detects operations occurring at points in the file-system name space where rules have been defined, creates symbolic links among objects as specified by the rules. This link driver can be inserted in the file-management or other modules of conventional operating systems without extensive modifications, perhaps even as a third-party device driver. Because the link driver creates and handles the links within the file system itself, the symbolic links are transparent to all application programs that access files and directories, and even to other levels of the file system itself. That is, an application accesses a linked file or directory with exactly the same mechanisms with which it accesses any file or directory; with no change whatsoever to the application code, and without complying with any additional standard or protocol. A rules component or query processor, either within the file-system link driver or located separately, stores and accesses the data required to carry out the rules. The rules defining a given set of dynamic symbolic links can be satisfied directly by the link driver; that is, the link driver can directly execute and satisfy the rule. Alternatively, the rule can be forwarded to any supported query processor. For example, Microsoft.RTM. WindowsNT.RTM. 5 supports full content indexing on all files, and its Windows content-indexing component can be the target of the dynamic symbolic link rule. The content-indexing component can receive the query, execute it, and return the result to the link driver, which can use the result to execute the file or directory operation.

Brief Summary Text (11):
The utility program receives from a user a request to associate certain file-system objects. The utility requests the user to specify a location in the file system name space for the association, and to specify a rule, association, relation, or query (these terms are interchangeable) for the insertion point. The utility then installs that rule at the insertion point. A subsequent request from a user application program--or, equivalently, from the system browser--for a file-system object such as a file or a directory at the insertion point invokes the file-system component to call the rules component to determine which objects at what actual locations fit the rule for that insertion point. The file system creates the appropriate links for objects that satisfy the rule, effectively plugging their names or other identifiers into the file system at the insertion point. The file system thereafter returns the objects to the application program transparently, exactly as though they actually existed at the insertion point. The next time the application--or any other application--requests the object, the rules are executed again, and new links are created; that is, the links themselves do not exist between accesses, but only the rules for creating them. Therefore, should a user delete a linked object or move it to a different location during an access, the object might no longer exist or satisfy the rule, and a subsequent access to the same insertion point will simply not

create a link to that object. The links thus require no maintenance or administration whatsoever, and there is no time interval during which they are incorrect or obsolete.

Detailed Description Text (4):
This section provides a brief, general description of a suitable computing environment in which the invention may be implemented. The invention will hereinafter be described in the general context of computer-executable program modules containing instructions executed by a personal computer (PC). Program modules include routines, programs, objects, components, data structures, etc. that perform particular tasks or implement particular abstract data types. Those in the art will appreciate that the invention may be practiced with other computer-system configurations, including handheld devices, multiprocessor systems, microprocessor-based programmable consumer electronics, network PCs, minicomputers, mainframe computers, and the like. The invention may also be practiced in distributed computing environments where tasks are performed by remote processing devices linked through a communications network. In a distributed computing environment, program modules may be located in both local and remote memory storage devices.

Detailed Description Text (5):
FIG. 1 employs a general-purpose computing device in the form of a conventional personal computer 20, which includes processing unit 21, system memory 22, and system bus 23 that couples the system memory and other system components to processing unit 21. System bus 23 may be any of several types, including a memory bus or memory controller, a peripheral bus, and a local bus, and may use any of a variety of bus structures. System memory 22 includes read-only memory (ROM) 24 and random-access memory (RAM) 25. A basic input/output system (BIOS) 26, stored in ROM 24, contains the basic routines that transfer information between components of personal computer 20. BIOS 24 also contains start-up routines for the system. Personal computer 20 further includes hard disk drive 27 for reading from and writing to a hard disk (not shown), magnetic disk drive 28 for reading from and writing to a removable magnetic disk 29, and optical disk drive 30 for reading from and writing to a removable optical disk 31 such as a CD-ROM or other optical medium. Hard disk drive 27, magnetic disk drive 28, and optical disk drive 30 are connected to system bus 23 by a hard-disk drive interface 32, a magnetic-disk drive interface 33, and an optical-drive interface 34, respectively. The drives and their associated computer-readable media provide nonvolatile storage of computer-readable instructions, data structures, program modules and other data for personal computer 20. Although the exemplary environment described herein employs a hard disk, a removable magnetic disk 29 and a removable optical disk 31, those skilled in the art will appreciate that other types of computer-readable media which can store data accessible by a computer may also be used in the exemplary operating environment. Such media may include magnetic cassettes, flash-memory cards, digital versatile disks, Bernoulli cartridges, RAMs, ROMs, and the like.

Detailed Description Text (6):
Program modules may be stored on the hard disk, magnetic disk 29, optical disk 31, ROM 24 and RAM 25. Program modules may include operating system 35, one or more application programs 36, other program modules 37, and program data 38. A user may enter commands and information into personal computer 20 through input devices such as a keyboard 40 and a pointing device 42. Other input devices (not shown) may include a microphone, joystick, game pad, satellite dish, scanner, or the like. These and other input devices are often connected to the processing unit 21 through a serial-port interface 46 coupled to system bus 23; but they may be connected through other interfaces not shown in FIG. 1, such as a parallel port, a game port, or a universal serial bus (USB). A monitor 47 or other display device also connects to system bus 23 via an interface such as a video adapter 48. In addition to the monitor, personal computers typically include other peripheral output devices (not shown) such as speakers and printers.

Detailed Description Text (10):
FIG. 2 shows a specific example of a multi-level hierarchical file structure 200 of the type implemented by many conventional computer operating systems, such as Microsoft Windows NT. File structure 200 is organized as a tree having a root node 201 containing a root directory. The root directory contains some files 203, and two subdirectories 204 and 205. The terms `directory` and `subdirectory` are equivalent for the present purpose. The term `file object` is used herein to denote either a directory or a file, and can also be applied to any other resource that can be stored in or referenced by a file system, such as a named pipe or a mailslot. Directory 204 contains two further subdirectories 207 and 208, and therefore defines another node 209. Directory 210 at node 211 contains files and subdirectory 213 at node 214.

Directory 213 contains files 215 and yet another directory 216. The dotted lines at node 217 indicate further files and subdirectories to any desired level. Directory 218, located at node 209, may also contain further objects at node 219. Directory 205 is shown as containing files 220 at its node 221, and also contains an insertion point 230. An insertion point is a type of file object for attaching a rule according to the invention.

Detailed Description Text (12):
The reparse information for effecting the invention includes a representation of a rule or query 231 that defines certain characteristics of the files or other objects for which dynamic links are to be constructed. These files, directories, etc. are actually located in other directories scattered arbitrarily throughout tree 200. For example one or more of the files 212, 216, and the directory 218, located in directories 210, 213, and 204 might satisfy the conditions of rule 231. Dynamic links, shown as lines 232-234, then cause such files to appear to reside in a directory located at insertion point 230. Continuing the previous example, if the name of root directory is "C:.backslash.y ", the name of directory 205 is "LegalDocs", and the name of the faux directory at point 230 is "Sue_Donym", then a request to list the contents of directory "C:.backslash.LegalDocs.backslash.Sue_Donym" returns the names, attributes, and other information for a set of phantom files and directories 235 that are actually located at other places throughout the file structure 200. The requesting program, however, can deal with them as though they were in fact located at the insertion point 230. For example, a user application program can open, modify, move, or delete a file. Any action upon such a file or directory object takes place upon the requested object at its actual location, transparently to the application. When the requesting program closes its access, links 232-234 go away.

Detailed Description Text (13):
FIG. 3 shows a system 300 for creating and manipulating dynamic links such as 232-234, FIG. 2. Almost all operating systems such as 35, FIG. 1, include a component known as a file system, file processor, file manager, or similar component. In the example WindowsNT operating system, input/output manager (NTIO) 310 fulfills this overall function as well as other functions not relevant to the present invention; therefore, block 310 is referred to herein as the file system or file manager. (By way of background, the WindowsNT 5.0 NTIO component of this embodiment controls all storage-related I/O and provides the file system interface, but it calls file-system drivers to handle the actual file operations. That is, NTIO does not handle I/O in and of itself. Instead, it calls the appropriate client file-system drivers, called NTFS in WindowsNT.) OS components such as manager 310 typically run in a kernel mode that protects them from user-level programs. A user-level association utility 320 defines which file objects will be linked to which insertion points 230, FIG. 2. The definition can be provided by a user employing input/output (110) devices such as keyboard 40 and display 47; alternatively, some other program can provide or modify the definition. Any application program 330 may request a particular file service in an entirely conventional manner by issuing a standard API (application program interface) to I/O manager 310. Manager 310 processes the API and ultimately issues a request to a storage device 330 that contains data for satisfying the request. The 110 manager processes this data, and either returns it to requesting program 310 or uses it to produce some other reply. In some cases, such as an incorrect syntax in the request, manager 310 turns the request around itself, producing a reply without ever reaching a storage device.

Detailed Description Text (14):
Like most modern file systems, manager 310 has a layered internal structure and employs pluggable components for adding and modifying functionality. Conventional device drivers 311, for example, translate requests in a common internal format into commands and data understood by specific kinds of storage devices, and reformat return data from the devices back into the common format. File system drivers 312 and similar modules perform intermediate functions such as error checking and file sharing. Filter drivers 313 form a newer class of file-system component. They detect a designated condition, flag, or characteristic at a specific I/O-manager level, and intercept a request or returned data. Filter drivers are drivers that receive all I/O requests sent to the target file system and at each level have an opportunity to satisfy the request, fail the request, modify the request, or issue additional requests to satisfy the request that they have received. A filter driver can intercept a file operation between any two layers traveling from an application program downward through the file manager to a physical storage device, and again on the way back from the device to the program. The filter driver then performs a function and can place the request or data back into the manager. Filter drivers are transparent to the operation of I/O manager 310. Conventional filter drivers

perform functions such as virus detection, data encryption, and hierarchical storage management. Filter drivers can invoke the services of other modules in the file system, other components of the operating system, and any other program at any level.

Detailed Description Text (15):
System 300 couples a dynamic-link (DL) filter driver 313 to I/O manager 310 for processing dynamic links during application-program accesses to the file system. Driver 313 also connects to a kernel-mode client 341 of a database management subsystem (DBMS) 340. Clients of this type provide a local user interface to a larger DBMS server located either in the same computer or at a remote facility. In this example, DBMS server 342 operates in the user space of the local computer 20, FIG. 1, and connects to client 341 via a conventional block of shared memory 343. A client for use in the invention entails a relatively small and easily written code module for invoking database queries in response to inputs from DL driver 313. Placing the client in the kernel space allows more direct communication with driver 313.

Detailed Description Text (17):
File indexer 345 is a facility found in the Enterprise Edition of WindowsNT. Its normal function is to continually update a relational database 346 holding certain characteristics of all the objects stored in file structure 200. That is, columns of database 346 can hold the type, size, date, and other attributes of files and directories. Database 346 can also employ a database to index file contents: words in a document, names in a spreadsheet, and so forth. In system 300, search engine 344 parses a query at an insertion point, and then DBMS server 342 redirects the query to database 346 of file indexer 345.

Detailed Description Text (19):
FIG. 4 is a flowchart of a method 400 for implementing association utility 320, FIG. 3. A user (or another program) requests at block 410 that dynamic symbolic links be defined. Block 420 receives the user's designation of a point such as 230 in the directory structure 200, FIG. 2, for insertion of the links. Block 430 requests the services of I/O manager 310 to create a reparse point at the designated insertion point. The illustrative WindowsNT file manager creates an empty directory with its reparse attribute set. Block 440 receives the user's definition of a rule for specifying which files should appear at the insertion point. In this implementation, the rule has the form of a query in the wellknown SQL query language. For example, the query

Detailed Description Text (24):
FIG. 5 diagrams a method 500 for creating and manipulating dynamic symbolic links that have been defined by method 400, FIG. 4.

Detailed Description Text (25):
Block 510 initiates method 500 in file system 310, whenever it receives a request for a file-system service or operation form a user-level application program or from some other source. This description uses a request for a directory listing as an illustrative example of a file operation. Block 520 traverses file structure 200 searching for the requested file object. If block 521 does not encounter a reparse point, block 530 in the file system obtains the object, using the appropriate file-system drivers and device drivers shown in FIG. 3. Block 531 returns the object to the requesting application. For a directory request, the actual form of the object is a list of names, attributes, locations, and other information for each file in the directory. Where the directory is a conventional one, the file information is that of the actual files in the entries for that directory.

Detailed Description Text (26):
If block 521 encounters a reparse point having a return code indicating that it is an insertion point for dynamic links, then DL driver 313 intercepts the STATUS_REPARSE error code in block 522. (Without this intercept, any error code would be passed all the way back to the requester.) Block 540 sends the query text from the reparse point to client 341 of DBMS subsystem 340, which connects DBMS server 342 and passes the query to it for parsing and execution in a conventional manner. Filter driver 313 matches the "remainingName" against the list of files it gets from the DBMS, and constructs the new name of the file to be opened. Although server 342 could execute the query on a database that it manages, block 541 instead redirects the query to file database 346. Block 550 indicates the action of indexer 345 in continually indexing all the files in structure 200 asynchronously to the remainder of process 500, so as to maintain file database 346 current at all times. Method 500 can, however, employ

other mechanisms for providing a current database or some other form of file information for creating dynamic links. For example, the file-structure directories themselves can form a searchable database. More generally, the invention can use any form of query processor, and can alternatively employ rule processors of other kinds, such as those found in expert systems.

Detailed Description Text (27):
Blocks 560 return database information for those files in structure 200 that satisfy the query. Block 561 opens a conventional cursor on the database. For each database row that satisfies the query, block 562 increments the cursor to point to that row in turn as the current row. Block 563 fetches the current row. Block 564 reformats the information in the row into proper form if required, and block 565 stores the information in the file object. For a directory-listing request, each row represents a file, and the columns contain file information, as described above. One of the file-information items is the actual location of the file in file structure 200. This location thus is a link to the file. When all rows have been processed, block 562 causes block 530 in I/O manager 310 to get the file object. Block 531 then returns the object to the requesting program 330 in the normal manner. In many file systems, block 531 returns a handle by which the requesting program can access a file object, rather than returning the object itself. Some operations, such as the directory listing used herein as an example, then employ the handle to obtain and return one file at a time through blocks 530 and 531 as the requesting program issues successive APIs for them, rather than returning all the information at once.

Detailed Description Text (28):
In this way, conventional application programs request and receive file objects in an entirely transparent manner, without any modification whatsoever. For example, an application program requests a directory listing using a conventional API for that purpose. The file system returns it via another standard API. The format of the listing is conventional, and the application can then open one of the files in the listing in its usual way. Method 500 at this point reads the file location at block 520, which now points to the actual location of the file in structure 200, rather than to the phantom directory at insertion point 230. File-system block 530 retrieves the file from its actual location, and block 531 returns it to the requesting application, all in a conventional manner. The only difference is that the file location information now forms a dynamic link to the actual location, rather than to the phantom directory at the insertion point that the user program thought it had requested. Any operation that application program 330 performs on the file, including modification, renaming, or deletion, is carried out on the real file at its actual location. Whenever the application releases an access via a standard file-system API, its file object disappears as it normally would. Therefore, the dynamic link merely disappears. The next request, by that application or by any other program, to insertion point 230 reexecutes method 500. Because block 550 continually updates the file-structure database, the links dynamically constructed by blocks 560 from their defining query might well differ from the links built during the previous request.

Detailed Description Text (29):
FIG. 6 is a flowchart 600 showing how method 500 retrieves dynamically linked files for an application program. Block 610 receives a request to, for example, open a file at block 610. Block 610 functions in exactly the same way as block 510, whether or not the requested file has a dynamic link. Block 620 follows the path of the directory returned by block 531, in exactly the same manner as block 530. The only difference is that the path to the actual location of the file was not placed in the directory by the file manager itself, but rather by DL filter driver 313. Because the file location actually exists in the file structure, it is treated like any other file, directory, or even another insertion point. Block 630 gets the file from its actual location just as block 530 would, and block 631 returns the file to the requesting program in the same way as block 531, FIG. 5. All of these blocks execute in the file manager 310 in a manner entirely transparent to the application program. That is, the application program need have no knowledge of the dynamic link, and need not request or receive the file any differently than a normal file.

Detailed Description Text (30):
Although dynamic symbolic links as described above are transparent to conventional application programs in almost all respects, there are a few differences. For example, the existence of multiple files having the same name is possible when the files are located in different actual directories. If such identically named files satisfy a query, then a request by an application

file via an insertion point for a file having that name will produce an ambiguity. This issue could be resolved in any of several ways, such as by appending unique identifiers to the name of the different actual files. Directories at insertion points are read-only, so that no new files can be created in these directories; again, this facility can be provided if desired.

Detailed Description Text (31):
Many variations and alternatives within the scope of the following claims will appear to those skilled in the art. In particular, although the above description locates much of the invention in the kernel of an operating system, any component of the invention can be located in any defined level or space in the programming environment of a computer. Also, the terms "operating system," "file manager," "indexer," and so forth must be construed broadly. Software or hardware components having different names and overall functions may also serve the present invention. It must also be remembered that the methods defined below can be performed in any temporal order, except where specifically indicated in the claims.

CLAIMS:

1. A method executed on a programmable computer for managing a file structure of file objects in a file system, the method comprising:

defining a rule specifying a desired characteristic for a set of the file objects in the file system;

designating a point in the file structure as an insertion point;

persistently associating the rule with the insertion point;

receiving a file-system request directed to the insertion point;

constructing a set of dynamic links to the set of the file objects that satisfy the rule; and

returning the dynamic links in a response to the request.

3. A method according to claim 1 wherein the file structure is a multi-level hierarchical structure.

7. A method according to claim 1 wherein the receiving step comprises:

receiving in the file system a name indicating one of a number of points in the file structure;

detecting that the one point is the insertion point.

8. A method according to claim 1 wherein the constructing step comprises:

searching the file structure for a set of file objects that satisfy the rule.

9. A method according to claim 8 wherein the file objects in the structure have respective locations within the structure, and wherein the constructing step further comprises

returning the actual location of each of the files in the set as a dynamic link for that file.

11. A method according to claim 1 further comprising indexing the file objects in the file structure asynchronously with respect to the remaining steps.

12. A method according to claim 1 wherein the indexing step includes building a database of the file objects in the file structure.

14. A programmable digital computer for executing application programs that access file objects located at multiple points in a file structure of a file system, the computer comprising:

storage means for storing the file objects;

input-output means for receiving a rule and for designating one of said multiple points as an insertion point, wherein the rule is persistently associated with the insertion point;

query processor means for determining which of the file objects at any of the multiple points satisfy the rule;

file processor means for managing the file structure;

dynamic link driver means coupled to the file processor means and to the query processor for detecting an access request directed to the insertion point, and in response thereto for constructing a set of dynamic links from the insertion point to those of the file objects that satisfy the rule.

21. A method executed on a computer for managing a file structure of file objects at multiple different locations, comprising:

receiving a request for a file object at one of the locations;

detecting that the one location is an insertion point;

accessing a rule persistently associated with the insertion point;

determining the locations of the file structure of a set of file objects that satisfy the rule;

constructing a set of dynamic links from the insertion point to each of the locations of the file objects in the set;

returning the dynamic links as a response to the request.

23. A method according to claim 21 wherein the file structure is a multi-level hierarchical structure.

25. A method according to claim 24 wherein determining the locations in the file structure of a set of file objects that satisfy the rule comprises searching a database for file objects that satisfy the query.

27. A method according to claim 25 further comprising indexing the file structure so as to construct the database.

29. A method according to claim 21 wherein the rule designates certain characteristics of the set of file objects, and wherein constructing the dynamic links comprises:

searching the file structure for the set of the file objects having the certain characteristics;

constructing a separate dynamic link for each of the file objects in the set to the insertion point.

32. A method according to claim 31 wherein accessing the one file object comprises:

receiving a request for the one file object directed to the insertion point;

reading the actual location of the one file object in the file structure from the dynamic link for the one file object;

returning the one file object from its actual location.

33. A programmable digital computer for executing application programs that access file objects located at multiple points in a file structure of a file system, the computer comprising:

a file manager for receiving file-object requests from the application program and for returning requested file objects thereto;

a link driver coupled to the file manager for intercepting requests directed to a predefined insertion point in the file structure and for fetching a rule persistently associated with the insertion point;

a subsystem coupled to the link driver for determining a set of the file objects at any of the multiple points in the file structure that satisfy the rule,

the link driver further returning the set of file objects as the requested file objects.

37. A computer according to claim 33 further including an indexer for constructing and continually updating a database of the file objects in the file structure.

39. A method executed on a computer for creating dynamic symbolic links in a file structure of file objects at multiple different locations in a file system, comprising:

receiving a designation of a desired location for the set of file objects;

creating an insertion point at the desired location;

receiving a rule specifying a desired characteristic of the file objects in the set, wherein each file object in the set satisfies the rule;

receiving a rule specifying the file objects in the set;

persistently associating the rule with the insertion point.

44. A method according to claim 39 wherein the rule is stored in the file structure at the insertion point.

Previous Doc Next Doc Go to Doc#

L14: Entry 8 of 8 File: USPT Nov 3, 1998

DOCUMENT-IDENTIFIER: US 5832527 A
TITLE: File management system incorporating soft link data to access stored objects

Abstract Text (1):
A file management system has a disk access unit for converting the relative position of a block in a file system into a physical position and accessing data at the position on a disk. The file management system includes a file entity operation unit, a file system manager access unit, and a file management table access unit. The file entity operation unit updates free block data in the file system when creating and deleting a file entity. The file system manager access unit lets the file entity operation unit work with no regard to the medium structure of a file system manager of the file system. The file management table access unit has an entry operation unit for allocating and releasing file management table entries and an entry access unit for referring to and updating the contents of the file management table entries. The entry access unit has a function of extracting soft link data out of any one of the file management table entries. The file management system allocates no data storage block to the soft link data, to thereby improve the space efficiency of the file system. The file management system refers to the contents of the symbolic link without referring to a data storage area of the file system, to thereby decrease the number of I/O operations during a pathname resolution process by the number of symbolic links contained in a path name to be solved.

Brief Summary Text (3):
The present invention relates to a file management system, and particularly, to a file management system capable of creating symbolic links for specifying files in a computer system.

Brief Summary Text (5):
A file management system manages file systems each composed of, for example, a file system manager for managing the name, size, conditions, and free block data of the file system, a file management table for managing the attributes and allocation data of files, and a data storage area for storing the contents of the files.

Brief Summary Text (6):
The file management system may create a symbolic link in a file system by allocating a file management table entry and at least a block (an I/O unit with respect to a disk) of the data storage area to the symbolic link, similar to creating a regular file. The file management table entry points the data storage block allocated to the symbolic link. The symbolic link contains a path name (soft link data), which is stored in the allocated data block. A file name and a file management table entry identifier for the symbolic link are registered in a proper directory to access the symbolic link.

Brief Summary Text (7):
In this way, the conventional file management system stores the contents of each symbolic link in a data storage block. The contents of each symbolic link are usually a path name composed of several tens of bytes, and therefore, storing the symbolic link contents deteriorates the space efficiency of the data storage area.

Brief Summary Text (8):
A pathname resolution process including the symbolic link requires a large number of I/O operations compared with that of a regular file (not including the symbolic link), to deteriorate the performance of the file management system. To narrow the gap between the pathname resolution including the symbolic link and the pathname resolution not including the symbolic link (regular file) in connection with the pathname resolution, the number of I/O

operations required by the symbolic link must be reduced.

Brief Summary Text (13):
According to the present invention, there is provided a file management system comprising a disk access unit for converting the relative position of a block in a file system into a physical position and accessing data at the physical position in a disk; a file entity operation unit for updating free block data in the file system when creating and deleting a file entity; a file system manager access unit for letting the file entity operation unit access a file system manager of the file system without knowing the data structure of the file system manager; and a file management table access unit having an entry operation unit for allocating and releasing file management table entries and an entry access unit for referring to and updating the contents of each of the file management table entries, the entry access unit having a function of extracting soft link data out of any one of the file management table entries.

Brief Summary Text (14):
The file management system may further comprise a higher unit; the entry access unit may be directly or indirectly used by the higher unit, to provide the contents of a file management table entry corresponding to an entry identifier specified by the higher unit irrelevant to the data structure of the file management table entry, data irrelevant to the data structure of the file management table entries may be received, the received data according to the data structure of the file management table entries may be processed, and one of the file management table entries according to the processed data may be updated; and the entry operation unit may be used by the file entity operation unit, to prepare a file management table entry and notify the higher unit of an entry identifier unique to the prepared file management table entry, as well as releasing a file management table entry corresponding to an entry identifier provided by the higher unit.

Brief Summary Text (15):
The higher unit may comprise an application program interface, pathname resolution unit, and hard link data operation unit; and the entry access unit may be directly or indirectly used by the application program interface, the pathname resolution unit, and the hard link data operation unit. The file management system may further comprise a data storage area access unit having a directory access unit for referring to, updating, deleting, and adding directory entries and a regular file access unit for referring to and updating data of regular files; the hard link data operation unit may use the directory access unit and the entry access unit, to refer to, update, add, and delete hard link data while maintaining the correspondence of the hard link data in a directory and a file management table entry; the pathname resolution unit may use the directory access unit and the entry access unit, to specify a file according to a given path name, hard link data, and soft link data; and the file entity operation unit may allocate and release file management table entries while correctly maintaining a relationship between file system free block data in the file system manager and allocation data in the file manager.

Brief Summary Text (16):
The file system manager of each file system may hold data for indicating whether the file system is new or old, and the file management system may have logic for determining a position to store soft link data according to the data indicating whether the file system is new or old, to thereby support old and new media.

Brief Summary Text (18):
The soft link data may be a path name that is the contents of a symbolic link serving as unit to specify a file in the file system. The file management method may be applied to manage data of files. The file management method may be applied to manage circulation files.

Drawing Description Text (3):
FIG. 1 illustrates a symbolic link managed by a file management system according to the present invention;

Drawing Description Text (5):
FIGS. 3A and 3B are flowcharts showing steps of creating a file according to a symbolic link process achieved by the file management system of the present invention;

Drawing Description Text (6):
FIGS. 4A and 4B are flowcharts showing steps of deleting a file according to the symbolic link process;

Drawing Description Text (7):
FIGS. 5A to 5C are flowcharts showing pathname resolution steps according to the symbolic link process;

Drawing Description Text (8):
FIGS. 6A and 6B show file management table entries of a fixed length data structure managed by the file management system of the present invention;

Drawing Description Text (9):
FIGS. 7A and 7B show file management table entries of a variable length data structure managed by the file management system of the present invention;

Drawing Description Text (15):
FIG. 13 shows steps of creating a symbolic link according to the file management system of the present invention;

Drawing Description Text (16):
FIG. 14 shows steps of deleting a symbolic link according to the file management system of the present invention;

Drawing Description Text (17):
FIGS. 15A and 15B are flowcharts showing steps of creating a file according to another symbolic link process achieved by the file management system of the present invention;

Drawing Description Text (18):
FIGS. 16A and 16B are flowcharts showing steps of deleting a file according to the symbolic link process;

Drawing Description Text (19):
FIGS. 17A to 17C are flowcharts showing pathname resolution steps according to the symbolic link process;

Drawing Description Text (20):
FIG. 18 shows the logic structure of a file system managed by the file management system of the present invention;

Detailed Description Text (2):
A symbolic link will now be briefly explained.

Detailed Description Text (3):
A unique file name is allocated to a file in a directory. Each directory is a file for managing many file names. A file hierarchy is composed of directories and other files. Any file name may be managed by a plurality of directories. Namely, any file may be specified with a plurality of path names each uniquely identifying the file. The file hierarchy may include, according to UNIX (registered trade name) specifications, directory files, regular files, block special files, character special files, FIFO special files, and symbolic links.

Detailed Description Text (4):
FIG. 1 shows an example of a symbolic link managed by a file management system according to the present invention. A file represented with a circle has a file name "bin." The file "bin" is controlled by two directories "/sys" and "/usr." Accordingly, the file "bin" is specifiable by any one of two path names "/sys/bin" and "/usr/bin."

Detailed Description Text (5):
To control files under a directory and specify any file with a path name, the directory must control a relationship between the name and entity of each file. A symbolic link may be prepared for any file. The symbolic link is a file and contains a path name to specify the file for which the symbolic link has been prepared.

Detailed Description Text (6):
In FIG. 1, a square represents a symbolic link. Characters in the square are the contents, i.e., a path name of the symbolic link. When a path name of "/sys2/bin" is entered, the system recognizes that the "/sys2" represents the symbolic link, excludes the path name of the symbolic link from the entered path name to provide "/bin," and adds the contents "/usr" of the symbolic link thereto to provide a path name of "/usr/bin" to specify the file "bin." The file "bin" of FIG. 1 is specifiable with any one of the three path names "/usr/bin," "/sys/bin," and "sys2/bin."

Detailed Description Text (7):
When a relationship between the name and entity of a file is added to a directory, a relationship called a hard link (or a link) is established between the directory and the file. On the other hand, a relationship between the contents of a symbolic link and a file is called a soft link (or a symbolic link). The contents of a symbolic link are determined by a user when creating a file, and there is no means to change the contents.

Detailed Description Text (8):
Each file system is composed of a file system manager, a file management table, and a data storage area. The file system manager manages the name, size, conditions, and free block data of the file system.

Detailed Description Text (10):
The file management system may create a symbolic link in any file system by allocating a file management table entry and at least a block (an I/O unit with respect to a disk) of the data storage area to the symbolic link, similar to creating a regular file. The file management table entry points to the allocated data storage block. The symbolic link contains a path name, which is stored in the allocated data storage block. A combination of a file name of the symbolic link and an identifier indicating the file management table entry is called a directory entry, which is registered in a proper directory so that the symbolic link can be referred to.

Detailed Description Text (17):
If the file is a symbolic link with i=N and if the symbolic link itself is an object to be processed (for example, deleted), the pathname resolution process ends. If the symbolic link is not the object to be processed, a data storage block allocated to the symbolic link is located according to allocation data in the file management table entry, to refer to the contents of the symbolic link. The first to "i"th file names are removed from the path name, and the remnant is added to the contents of the symbolic link, to provide a path name, which is again processed from the step 1.

Detailed Description Text (18):
If the file is not a directory nor a symbolic link and if i=N, the pathname resolution process ends. If the file is not a directory nor a symbolic link and if i<N, it is notified that the specified file has not been found.

Detailed Description Text (19):
Storing the contents of a symbolic link in the data storage area has some problems. For example, contents of a symbolic link is a path name, which is usually composed of several tens of bytes. A minimum allocation unit of the data storage area is equal to an I/O block. According to some I/O architecture, the block is made of several thousands of bytes. Allocating such a large block to a small symbolic link deteriorates the space efficiency of the data storage area.

Detailed Description Text (20):
As mentioned above, a path name involving a symbolic link needs more I/O operations during the pathname resolution process compared with a path name involving no symbolic link, to drastically deteriorate the performance of the system. To narrow the gap between the path names with and without the symbolic link during the pathname resolution process, the number of I/O operations required by the symbolic link containing path name must be reduced.

Detailed Description Text (22):
FIG. 2 is a block diagram showing the file management system. The system includes an application program interface (API) 1, a pathname resolution unit 2, a hard link data operation

unit 3, a file entity operation unit 4, a data storage area access unit 5, a file management table access unit 6, a file system manager access unit 7, a disk access unit 8, and a disk 9.

Detailed Description Text (23):
A file system managed by the file management system is an array of blocks and is composed of a file system manager, a file management table, and a data storage area. The file management system informs the disk access unit 8 of the relative position of a block in the file system. The disk access unit 8 converts the relative position into a physical position and accesses the position of the disk 9 for data.

Detailed Description Text (24):
The file system manager access unit 7 is used by the file entity operation unit 4 to update free block data in the file system when creating or deleting a file entity. The file system manager access unit 7 lets the file entity operation unit 4 work with no regard to the medium structure of the file system manager of the file system.

Detailed Description Text (25):
The file management table access unit 6 has an entry operation unit 61 for allocating and releasing file management table entries and an entry access unit 62 for referring to and updating the contents of the entries. The structure of the file management table is dependent on the file system. For example, the file system may employ fixed length entries or variable length entries, the entries being arranged in an array or a hash structure. Irrelevant to the structure of the file management table, the file management table access unit 6 provides an interface for the file manager.

Detailed Description Text (27):
The entry access unit 62 is directly or indirectly used by the pathname resolution unit 2, hard link data operation unit 3, and other units that use files through the API 1. Responding to an entry identifier provided by any higher unit, the entry access unit 62 notifies the higher unit of the contents of an entry corresponding to the entry identifier with no regard to the data structure of the entry. The entry access unit 62 receives data without knowing the data structure of the file management table entry, processes the data according to the data structure of the entry, and updates the entry according to the data.

Detailed Description Text (28):
According to the embodiment of FIG. 2, the entry access unit 62 has a function of extracting soft link data out of any entry. The file system manager may have data such as a file system name and a version to show whether the file system is new or old. In this case, new and old file systems are easily supported according to a logic that determines the storing position of soft link data according to whether the file system is new or old. It is not so easy to support new and old file systems having no such data for distinguishing new and old, but they may be supported if the data structure of the entries allows the tagging of age.

Detailed Description Text (30):
If the file management table of the old file system employs fixed length entries, the file management table of the new file system employs the same data structure, and in a file management table entry for a symbolic link, uses a section for data storage allocation data or a reserved section as a section for storing soft link data. Data such as the number of allocated blocks indicating that no data storage block is allocated must be left as it is. If a symbolic link is given and if no data storage block is allotted to the symbolic link, i.e., if the number of allocated data storage blocks is cleared to indicate that there is no allocated data storage block, soft link data related to the symbolic link are picked out from a file management table entry. If any data storage block is allocated to the symbolic link, the soft link data are extracted from the allocated data storage block. When creating a symbolic link, it is created as a new medium. In this way, it is possible to support the new and old file systems with entries of the new and old data structures.

Detailed Description Text (33):
The file entity operation unit 4 allocates and releases file management table entries while maintaining the correspondence between free space data in the file system manager and storage area allocation data in the file management table.

Detailed Description Text (34):

To improve operation speed, the file system manager access unit 7, file management table access unit 6, and data storage area access unit 5 may each have a function of caching part of a medium into a memory. In this case, no higher unit is required to determine whether data in the memory must be accessed or the medium must be accessed through the disk access unit 8. Namely, the units 7, 6, and 5 handle the data where it exists.

Detailed Description Text (35):
FIGS. 3A and 3B are flow charts showing steps of creating a file according to a symbolic link process achieved by the file management system of the present invention. FIGS. 4A and 4B are flowcharts showing steps of deleting a file according to the symbolic link process, and FIGS. 5A to 5C are flowcharts showing pathname resolution steps according to the same. The steps of creating and deleting a file of FIGS. 3A, 3B, 4A, and 4B clearly show differences from a conventional file creation process.

Detailed Description Text (37):
The step S106 allocates a file management table entry identifier and a file management table entry to the file to be created in the same file system as the parent directory. Step S107 determines whether or not it is the creation of a symbolic link. If it is the creation of the symbolic link, the flow jumps to step S109, and if not, step S108 is carried out. The step S108 allocates data storage blocks to the file. The step S109 prepares the file management table entry. Step S110 adds a directory entry, i.e., a combination of the file system name, file management table entry identifier, and file name of the created file to the parent directory. The step S102 is carried out to end the file creation process.

Detailed Description Text (39):
The step S206 deletes the directory entry from the parent directory. Step S207 retrieves a file management table entry corresponding to the deleted file in the file system and updates hard link data in the file management table entry. Step S208 determines whether or not the number of hard links is zero. If it is zero, the step S202 is carried out, and if it is not zero, step S209 deletes the file management table entry. Step S210 determines whether or not the file to be deleted is a symbolic link. If it is the symbolic link, the step S202 is carried out. If it is not the symbolic link, step S211 releases a corresponding data storage area, and the step S202 is carried out. The step S202 notifies a process result according to external specifications and terminates the process.

Detailed Description Text (42):
The step S306 extracts a file system name and a file management table entry identifier out of the retrieved directory entry and finds the file management table entry in the file system. Step S307 determines whether or not all file names in the path name have been solved. If all file names have been solved, step S309 is carried out, and if not, step S308 is carried out. The step S309 determines whether or not the file corresponding to the file management table entry is a symbolic link. If it is the symbolic link, step S310 is carried out, and if not, the flow goes to a normal end. The step S310 determines whether or not the symbolic link is an object to be processed. If it is the object to be processed, the flow goes to the normal end, and if not, step S311 is carried out. The normal end provides the file system name, file management table entry, and file type of the file specified by the entered path name, as well as the file system name and file management table entry of the parent directory.

Detailed Description Text (43):
The step S308 determines whether the file corresponding to the file management table entry is a directory or a symbolic link. If it is one of the directory and symbolic link, step S311 is carried out, and if not, the process goes to the abnormal end. The step S311 determines whether or not the file corresponding to the file management table entry is a symbolic link. If it is the symbolic link, step S314 picks out a path name in a soft link data section in the file manage entry, combines the same with the remnant of the path name now being solved, and provides a new path name to be solved. Thereafter, the step S301 is repeated. If the file corresponding to the file management table entry is not the symbolic link, step S312 prepares for accessing a data storage area according to data storage allocation data contained in the file management table entry. Step S313 selects the next file name to be solved in the entered path name, and the step S304 is repeated.

Detailed Description Text (44):
FIGS. 6A and 6B illustrate examples of fixed length file management table entries employed by

the file management system of the present invention, in which FIG. 6A is for files other than a symbolic link, and FIG. 6B for the symbolic link. In FIG. 6B, soft link data in the fixed length file management table entry are a path name that is the contents of the symbolic link and are composed of, for example, a character string of 512 bytes.

Detailed Description Text (45):
FIGS. 7A and 7B illustrate examples of variable length file management table entries employed by the file management system of the present invention, in which FIG. 7A is for files other than a symbolic link, and FIG. 7B for the symbolic link. In FIG. 7B, soft link data in the variable length file management table entry are the length and name of a path contained in the symbolic link and are composed of, for example, a character string of 512 bytes.

Detailed Description Text (46):
The file management table entries may have any structure to contain the soft link data. When the entries have a fixed length as shown in FIGS. 6A and 6B, they may deteriorate the space efficiency of the file management table but the file management table can be easily handled as an array of the entries. The data structure is dependent on system requirements. If the maximum length of a path name allowed by the system as an API function is short, the fixed length data structure will be appropriate. If the entry structure is originally of the variable length or if the maximum length of a path name allowed by the system is long, the variable length data structure of FIGS. 7A and 7B will be appropriate.

Detailed Description Text (54):
In FIG. 10, the file system manager manages the name and size of the file system, file management table allocation data, free block data, and a file management table entry identifier.

Detailed Description Text (55):
In FIG. 11, the file management table has, for example, a hash structure of 109 buckets, which are accessed with the file management table entry identifier as a key. An overflow area is used when an entry overflows from any bucket. The overflow area is managed by data in a superblock. Each bucket contains, for example, five blocks each storing at least one entry.

Detailed Description Text (58):
Security data include access permission, an owner identifier, and an owner group identifier. Hard link data include the number of hard links, and for each hard link, a set of (1) an entry identifier representing a hard link managing directory (a parent directory), (2) the name of a file system where the parent directory exists, and (3) a file name managed by the parent directory. Soft link data include a path name that is the contents of a symbolic link. A file other than the symbolic link does not have the sixth column for the soft link, so that it has five columns in total.

Detailed Description Text (59):
FIG. 13 shows an operation of the file management system of the present invention when creating a symbolic link, and FIG. 14 shows an operation of the file management system when deleting a symbolic link. For the sake of simplicity, a user space and the global service unit (GS) operate in the same processor module in FIGS. 13 and 14. The user space and GS may operate in different processor modules. The processor modules for operating the user space and GS are without knowing the spirit of the present invention.

Detailed Description Text (60):
FIGS. 15A and 15B are flowcharts that illustrate the steps of creating a file according to another symbolic link process achieved by the file management system of the present invention, FIGS. 16A and 16B are flowcharts showing steps of deleting a file according to the symbolic link process, and FIGS. 17A to 17C are flowcharts showing pathname resolution steps according to the symbolic link process.

Detailed Description Text (61):
The steps of creating a symbolic link will be explained with reference to FIGS. 13, 15A, and 15B. A user requests the local service unit (LS) to create a symbolic link "x/y" with "b" as its contents. Step SS1 examines whether or not each of the path names "x/y" and "b" is within a path name length allowed by the system, and prepares a message according to a protocol between the LS and GS, to request the GS to create a symbolic link.

Detailed Description Text (62):
The message transmitted from the LS to the GS includes a function code representing the request of creating a symbolic link, security data such as a user identifier and a group identifier, the path name "x/y" of the symbolic link to be created, and the path name "b" that is the contents of the symbolic link.

Detailed Description Text (63):
In step SS2, the LS sends the message prepared in the step SS1 to the GS and waits for a reply. Upon receiving the message from the LS, the GS uses a pathname resolution process to specify the name of a file system where a parent directory of the symbolic link to be created exists and a file management table entry identifier representing the parent directory. The details of this will be explained later.

Detailed Description Text (64):
Since the function code indicates a need to create a symbolic link, step SS3 obtains an exclusive right of the data storage area of the parent directory, so that the parent directory will not be updated by other threads, and at the same time, confirms that there is no directory entry involving the same file name as that of the symbolic link to be created in the parent directory. Step SS4 obtains an exclusive right of a file system manager of the file system where the parent directory exists and allocates a file management table entry identifier to the file to be created.

Detailed Description Text (65):
The entry identifier is an unsigned 4-byte integer, which is prepared whenever a file is created. The file system manager stores the latest entry identifier, which is incremented by one and is allocated to the file to be created. The incremented value is stored as the latest entry identifier in the file system manager. Since a frequently accessed portion of the file system manager is developed in a memory, both the memory and a medium are updated at this time. Once the file system manager is updated, the exclusive right of the file system manager is released.

Detailed Description Text (66):
Step SS6 prepares a file management table entry for the symbolic link according to the format of FIG. 12. Step SS7 divides the entry identifier by the number of buckets (109) and uses the remainder as a hash value to specify a bucket to store the file management table entry. An exclusive right of a file management table of the file system is obtained, and the file management table entry is stored in a free block in the specified bucket. Then, the medium is updated. If the bucket has no free block, the file management table entry is stored in an overflow area. Once the medium is updated accordingly, the exclusive right of the file management table is released.

Detailed Description Text (67):
Step SS9 prepares a directory entry, i.e., a set of the file name "y" of the created symbolic link, the file management table entry identifier, and a file type (symbolic link). At this time, the name of the file system is omitted because the file is created in the file system where the parent directory exists. The directory entry is added to the parent directory specified in the step SS3. When the medium is updated accordingly, the exclusive right of the parent directory is released. This embodiment includes the file type in the directory entry.

Detailed Description Text (69):
In this way, the present invention does not store soft link data related to the symbolic link in the data storage area, to thereby save and/or occupy a block (4096 bytes) of the data storage area. It is not necessary, therefore, to allocate a data storage block to the symbolic link nor change free space data in the file system manager. For these reasons, among others stated hereink, the management system will now be able to read with greater efficiency.

Detailed Description Text (70):
The steps of deleting a symbolic link will be explained with reference to FIGS. 14, 16A, and 16B. A user requests the LS to delete a file "x/y." Step SS21 examines whether or not the path name "x/y" is within the path name length allowed by the system and prepares a message according to the protocol between the LS and GS, to ask the GS to delete a hard link.

Detailed Description Text (72):
In step SS22, the LS sends the message prepared in the step SS21 to the GS and waits for a reply. Upon receiving the message from the LS, the GS employs the pathname resolution process to specify the name of a file system where the parent directory "x" of the symbolic link to be deleted exists and a file management table entry identifier representing the parent directory. The details of this will be explained later.

Detailed Description Text (74):
Step SS24 deletes the directory entry from the parent directory and releases the exclusive right of the data storage area of the parent directory. Step SS26 obtains an exclusive right of a file management table of the file system where the file to be deleted exists, and according to the file management table entry identifier, specifies a bucket and block where a corresponding file management table entry exists. The file name and the parent directory managing the file name are deleted from hard link data in the file management table entry, and the number of hard links is decremented by one in the hard link data. If the number of the hard links becomes 0, the file management table entry is deleted from the block. If the file management table entry is for a symbolic link, it has no space in the data storage area, so that the process of releasing the data storage area, i.e., the process of updating the free space data in the file system manager is not carried out.

Detailed Description Text (77):
In this way, the present invention does not store soft link data related to a symbolic link in the data storage area, so it is not necessary to release the data storage area nor update the free space data in the file system manager when deleting the symbolic link. The path name resolution process according to the symbolic link process is carried out in the GS, to retrieve directories and symbolic links contained in a given path name and specify the name of a file system where the file specified by the path name exists as well as the file management table entry identifier of the file.

Detailed Description Text (81):
Step SS34 determines whether or not the file in question is a symbolic link, whether or not the symbolic link is an object to be processed, and whether or not the parent directory must be updated. If i=N, the file is not a symbolic link, and the parent directory must be updated, or if i=N, the file is the symbolic link to be processed, and the parent directory must be updated, it is determined whether or not the user is authorized to update the parent directory, and the pathname resolution process ends. Step SS35 extracts a file system name and a file management table entry identifier, and according to these data, retrieves a file manger entry under the extracted file system name.

Detailed Description Text (84):
(II) If the file is a symbolic link, the first to "i"th file names are removed from the path name to be solved, the remnant is combined with a path name stored in the sixth column of the file management table entry, to provide a new path name to be solved, and the step SS31 is repeated.

Detailed Description Text (85):
(III) If the file is not the directory nor the symbolic link, the LS is informed that the file specified by the user has not been found.

Detailed Description Text (86):
In this way, the file management system of the present invention does not allocate a data storage block to soft link data, thereby improving the space efficiency of file systems. This effect of the present invention is more conspicuous in a system employing an I/O architecture with large blocks. The file management system of the present invention refers to the contents of a symbolic link without referring to the data storage area, thereby reducing the number of I/O operations during the pathname resolution process by the number of symbolic links appearing during the pathname resolution process.

Detailed Description Text (87):
FIG. 18 shows an example of the logic structure of a file system handled by the file management system of the present invention, and FIG. 19 shows a relationship between an MFD entry and a data area of the file system of FIG. 18. In FIG. 18, the file system is, for example, an SXO file system composed of "max" blocks.

Detailed Description Text (88):
In FIG. 18, blocks 1 to n form a file system free space management area MFS for managing free blocks in the file system. Blocks n+1 to m form a file entity management area, i.e., a file management table. This area is a collection of MFD entries for managing individual files. Each of the MFD entries involves (a) file type (regular file, directory, special file, symbolic link, or FIFO), (b) access time data, (c) security data, (d) data area allocation data (in case of the regular file and directory), (e) the contents of a symbolic link (in case of the symbolic link), and (f) hard link data.

Detailed Description Text (92):
As described in the underlined part of the above-provided algorithm by way of pseudo-code, the prior art stores a path name that is the contents of a symbolic link in the data area of the file system, and therefore, the stored path name must be referred to through I/O operations. On the other hand, the present invention stores the path name that is the contents of the symbolic link in an "inode" so that the present invention needs no I/O operation to refer to the contents of the symbolic link.

Detailed Description Text (101):
As explained above in detail, the file management system according to the present invention does not allocate data storage blocks to soft link data, to thereby improve the space efficiency of file systems. This effect is conspicuous in a system employing an I/O architecture with large blocks. In addition, the file management system of the present invention refers to the contents of a symbolic link without referring to a data storage area, to thereby reduce the number of I/O operations during the pathname resolution process by the number of symbolic links contained in a given path name.

Detailed Description Paragraph Table (1):
_____ parent inode .uparw. = NULL; /*firstly solve a current working directory if a relative path name is given*/ LOOP.sub.-- TOP: count the number of elements in the path name; /*specify a retrieval directory inode*/ if (absolute path name) { retrieval directory inode .uparw. = root directory inode .uparw.;} else if (parent inode .uparw. = = NULL) { retrieval directory inode .uparw. = current working directory inode .uparw.;} the number of processed path name elements = 0; while (the number of processed path name elements <= the number of path name elements and no error) { refer (I/O) to the inode and obtain data area allocation data; parent inode .uparw. = referred inode .uparw.; switch (file type) { directory: retrieve a directory entry in a directory data area with a file name as a key; specify an inode from the directory entry; break; symbolic link: refer (I/O) to a symbolic link data area and obtain a path name contained in the symbolic link. combine the obtained path name and the remaining path name together; go to LOOP.sub.-- TOP; break; default: if (the number of processed path name elements < the number of path name elements) { error detection; } } the number of processed path name elements += 1; } }

CLAIMS:

1. A file management system, comprising:

disk access means for converting a relative position of a block of a file system into a physical position and accessing data at said physical position in a disk;

file entity operation means for updating free block data in said file system when creating and deleting a file entity;

file system manager access means for defining a data structure of said file system using a file system manager and for providing access by said file entity operation means to said file system manager independently of the data structure; and

a file management table access unit to provide an interface to said file system manager access means, said file management table access unit having

entry operation means for allocating and releasing file management table entries, and

entry access means for referring to and updating the file management table entries and for extracting symbolic link data including a path name out of any one of said file management table entries.

2. A file management system as claimed in claim 1,

wherein said file management system further comprises identifier producing means for producing file system entry identifiers;

wherein said entry access means is directly or indirectly used by said identifier producing means, to provide the contents of a file management table entry corresponding to an entry identifier specified by said identifier producing means independently of the data structure of said file management table entry, to provide data independently of the data structure of said file management table entries is received, said received data according to the data structure of said file management table entries is processed, and one of said file management table entries according to the processed data is updated; and

wherein said entry operation means is used by said file entity operation means, to prepare a file management table entry and notify said identifier producing means of an entry identifier unique to said prepared file management table entry, as well as releasing a file management table entry corresponding to an entry identifier provided by said identifier producing means.

3. A file management system as claimed in claim 2,

wherein said identifier producing means comprises an application program interface, pathname resolution means for resolving path names, and link data operation means for determining a relative location of data; and

wherein said entry access means is directly or indirectly used by said application program interface, said path name resolution means, and said link data operation means.

4. A file management system as claimed in claim 3,

wherein said file management system further comprises data storage area access means for accessing a data storage area, said data storage area access means having

directory access means for referring to, updating, deleting, and adding directory entries, and regular file access means for referring to and updating data of regular files;

wherein said link data operation means uses said directory access means and said entry access means, to refer to, update, add, and delete link data while maintaining the correspondence of the link data in a directory and a file management table entry;

wherein said path name resolution means uses said directory access means and said entry access means, to specify a file according to a given path name, link data, and the path name included in the symbolic link data; and

wherein said file entity operation means allocates and releases file management table entries while correctly maintaining a relationship between file system free block data in said file system manager and allocation data in the file management table entries.

5. A file management system as claimed in claim 1, wherein said file system manager of each file system holds data for indicating whether said file system is new or old, and said file management system has logic for determining a position to store file access information according to the data indicating whether said file system is new or old, to thereby support old and new media.

6. A file management method of a computer, comprising the steps of:

storing symbolic link data including a path name related to a file, in a file management table pointed to by a directory entry of a computer file management system, the path name providing a symbolic link to specify another directory entry in the computer file management system; and

managing the file in the computer according to the symbolic link data.

9. A file management method as claimed in claim 6, wherein said computer file system is managed by a file management system comprising:

disk access means for converting the relative position of a block in said file system into a physical position and accessing data at said physical position in a disk:

file entity operation means for updating free block data in said file system when creating and deleting a file entity;

file system manager access means for defining a data structure of said file system using a file system manager and for providing access by said file entity operation means said file system manager independently of the data structure; and

a file management table access unit having

entry operation means for allocating and releasing file management table entries, and

entry access means for referring to and updating the file management table entries and for extracting symbolic link data out of any one of said file management table entries.

10. A file management method as claimed in claim 9,

wherein said file management system further comprises identifier producing means for producing file system entry identifiers;

wherein said entry access means is directly or indirectly used by said identifier producing means, to provide the contents of a file management table entry corresponding to an entry identifier specified by said identifier producing means independently of the data structure of said file management table entry, data without knowing the data structure of said file management table entries is received, said received data according to the data structure of said file management table entries is processed, and one of said file management table entries according to the processed data is updated; and

wherein said entry operation means is used by said file entity operation means, to prepare a file management table entry and notify said identifier producing means of an entry identifier unique to said prepared file management table entry, as well as releasing a file management table entry corresponding to an entry identifier provided by said identifier producing means.

11. A file management method as claimed in claim 10,

wherein said identifier producing means comprises an application program interface, path name resolution means for resolving path names, and link data operation means for determining a relative location of data; and

wherein said entry access means is directly or indirectly used by said application program interface, said path name resolution means, and said link data operation means.

12. A file management method as claimed in claim 11,

wherein said file management system further comprises a data storage area access means for accessing a data storage area, said data storage area access means having

directory access means for referring to, updating, deleting, and adding directory entries, and

regular file access means for referring to and updating data of regular files;

wherein said link data operation means uses said directory access means and said entry access means, to refer to, update, add, and delete link data while maintaining the correspondence of the link data in a directory and a file management table entry;

wherein said pathname resolution means uses said directory access means and said entry access

means; to specify a file according to a given path name, link data, and file access information; and

wherein said file entity operation means allocates and releases file management table entries while correctly maintaining a relationship between file system free block data in said file system manager and allocation data in the file management table entries.

13. A file management method as claimed in claim 9, wherein said file system manager of each file system holds data for indicating whether said file system is new or old, and said file management system has logic for determining a position to store the symbolic link data file access information according to the data indicating whether said file system is new or old, to thereby support old and new media.

Previous Doc Next Doc Go to Doc#